

UNIT 4

List

(2 Hours) — Data Structures and Algorithms (CACS201)

4.1 Introduction

A **List** is a linear data structure consisting of an ordered collection of elements, where each element (except the first) has a unique predecessor and each element (except the last) has a unique successor. Unlike a Stack or a Queue, a List does **not** restrict insertion and deletion to a fixed end — elements can be inserted, deleted, or accessed at **any position**.

Because of this flexibility, a List is considered the most general linear data structure. In fact, both the Stack and the Queue can be viewed as **restricted forms of a List**: a Stack allows operations only at one end, and a Queue allows insertion at one end and deletion at the other.

Structure	Where insertion/deletion is allowed
List	Anywhere in the sequence (any position)
Stack	Only at one end (the TOP)
Queue	Insertion at REAR, deletion at FRONT only

Note: A List that supports insertion and deletion at both ends (but not necessarily in the middle) is called a Deque (Double-Ended Queue) — a hybrid between a Stack and a Queue.

4.2 Static and Dynamic List Structure

4.2.1 Static List

A **static list** is implemented using an **array**. Its maximum size must be declared in advance, at compile time, and it occupies a fixed block of contiguous memory. The list grows or shrinks logically (through a length/count variable) but never beyond the pre-declared array capacity.

4.2.2 Dynamic List

A **dynamic list** is implemented using **linked allocation** — individual nodes are created and destroyed at run time using pointers/references, and are typically **not** stored in contiguous memory. The list can grow or shrink freely, limited only by the total memory available on the system.

Aspect	Static List (Array-based)	Dynamic List (Linked)
Memory allocation	Fixed, at compile time	Flexible, at run time
Memory layout	Contiguous	Scattered (linked via pointers)
Size	Fixed maximum	Grows/shrinks as needed
Random access	$O(1)$ via index	$O(n)$ — must traverse from start
Insertion/Deletion	Requires shifting elements	No shifting — only pointer updates
Memory efficiency	May waste unused declared space	Uses exactly what is needed

4.3 Array Implementation of Lists

In an array implementation, a List is represented by an array together with a variable (commonly called **LENGTH** or **COUNT**) that tracks how many positions currently hold valid data. Insertion and deletion at an arbitrary position require shifting the surrounding elements to keep the list contiguous.

Algorithm — Insert(item, pos)

1. If $LENGTH = MAX_SIZE$, print "List is full" and exit.
2. If $pos < 0$ or $pos > LENGTH$, print "Invalid position" and exit.
3. For $i = LENGTH$ down to $pos + 1$: set $LIST[i] = LIST[i - 1]$. (shift right)
4. Set $LIST[pos] = item$.
5. Set $LENGTH = LENGTH + 1$.

Algorithm — Delete(pos)

1. If $pos < 0$ or $pos \geq LENGTH$, print "Invalid position" and exit.
2. Set $item = LIST[pos]$.
3. For $i = pos$ to $LENGTH - 2$: set $LIST[i] = LIST[i + 1]$. (shift left)
4. Set $LENGTH = LENGTH - 1$.
5. Return item.

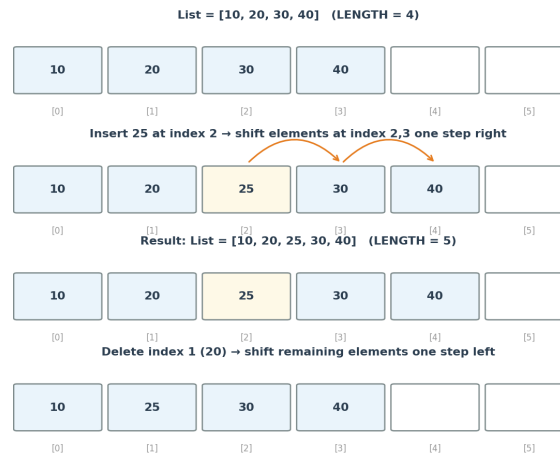


Figure 4.1 — Inserting 25 at index 2 (right shift) and deleting index 1 (left shift)

Complexity of Array-Based List Operations

Access by index: $O(1)$ — direct computation of memory address.

Search (unsorted): $O(n)$ — worst case scans the entire list.

Insert / Delete at the end: $O(1)$ — no shifting required.

Insert / Delete at the beginning or middle: $O(n)$ — up to n elements must shift.

Note: Because every mid-list insertion or deletion may require shifting up to n elements, array-based (static) lists are efficient for random access but comparatively costly for frequent insertions and deletions away from the end.

4.4 Queues as a List

A Queue can be understood as a **List with restricted access**: insertion is permitted only at one end (REAR) and deletion only at the other end (FRONT), whereas a general List permits both operations at any position. Viewed this way, the array-based Queue introduced in Unit 3 is simply an **array implementation of a List** in which the general `Insert(item, pos)` and `Delete(pos)` algorithms are specialised to two fixed positions:

General List Operation	Restricted to, in a Queue	Resulting Operation
<code>Insert(item, pos)</code>	<code>pos = LENGTH</code> (the end)	<code>enqueue(item)</code>
<code>Delete(pos)</code>	<code>pos = 0</code> (the beginning)	<code>dequeue()</code>

Implementing a queue as a restricted list this way still inherits the List's main weakness: deleting from position 0 requires shifting every remaining element one step left, exactly as in Unit 3's linear queue. This is precisely why the **circular queue** technique (REAR/FRONT wrap-around using modulo arithmetic) is preferred in practice — it avoids treating the queue as a plain shifting list and instead reuses freed array slots directly.

Note: This relationship also explains why a Deque (insertion/deletion at both ends) is easy to build from a List — it simply specialises Insert/Delete to $pos = 0$ and $pos = LENGTH$, without allowing arbitrary middle positions.

Key Takeaways

- ✓ A List is the most general linear structure — insertion and deletion are allowed at any position, not just at the ends.
- ✓ A Static List (array-based) has a fixed size and contiguous memory; a Dynamic List (linked) grows and shrinks at run time using pointers.
- ✓ Array-based lists give $O(1)$ indexed access but require shifting elements — $O(n)$ — for mid-list insertion or deletion.
- ✓ Both the Stack and the Queue can be viewed as Lists with restricted insertion/deletion positions: Stack restricts both to one end, Queue restricts insertion to REAR and deletion to FRONT.

— *End of Unit 4* —